Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Entwurfsmuster

Erzeuger und Strukturen

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

Verhalten

- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

Motivation

Der Entwurf objektorientierter Software ist schwer! Aber noch viel schwerer ist der Entwurf wiederverwendbarer objektorientierter Software, denn

- er muss spezifischen Anforderungen genügen, aber
- er muss allgemein genug sein, um zukünftigen Problemen und Anforderungen zu begegnen, und
- wir wollen eine Revision von Entwürfen vermeiden bzw. minimieren.

→ Vor der Fertigstellung eines Entwurfs versuchen wir, ihn wiederzuverwenden; eventuell ist dazu eine Änderung des Entwurfs nötig.

Motivation

Erfahrene Entwickler erstellen gute Entwürfe.

Was wissen die, was unerfahrene nicht wissen?

- Experten vermeiden es, jedes Problem von Grund auf neu anzugehen.
- Sie verwenden Lösungen wieder, die bereits zuvor erfolgreich eingesetzt wurden: *Entwurfsmuster*.

Eine gute, verständliche Einführung in das Thema gibt das Buch von Eric Freeman und Elisabeth Freeman: Entwurfsmuster von Kopf bis Fuß. O'Reilly-Verlag.

Entwurfsmuster

Erzeuger und Strukturen

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

Verhalten

- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

Motivation: Wir haben ein Software-Produkt erstellt, das an mehreren Stellen an die Bedürfnisse unserer einzelnen Kunden angepasst werden muss:

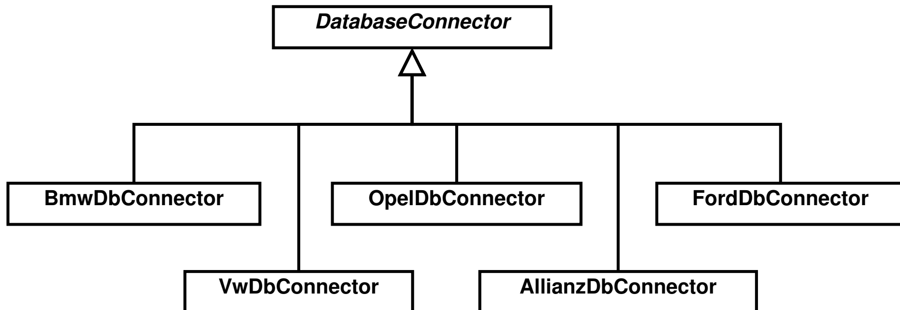
- Verbindung zur Datenbank → `DatabaseConnector`
- Format des Loggings → `Logger`
- Login-Prozess → `Authenticator`
- Verfahren zur Lastbalancierung → `LoadBalancer`

Obige Klassen sind Supertypen, die die Schnittstelle festlegen.

Für unsere Kunden BMW, Ford, Opel, VW, Deutsche Bank, Allianz, Provinzial und Ergo haben wir jeweils spezielle Subklassen implementiert.

Abstract Factory

- Erzeuge für jeden Kunden eigene Subklassen obiger abstrakter Supertypen und
- plaziere möglichst viel gemeinsamen Code in den Basisklassen.



Wie kann der Kunde hinterlegt werden `Kunde=Ergo`?

- In Konfigurationsdateien wie ini- oder properties-Datei ablegen.
- In C++ kann die Übersetzung durch Compiler-Direktiven gesteuert werden.

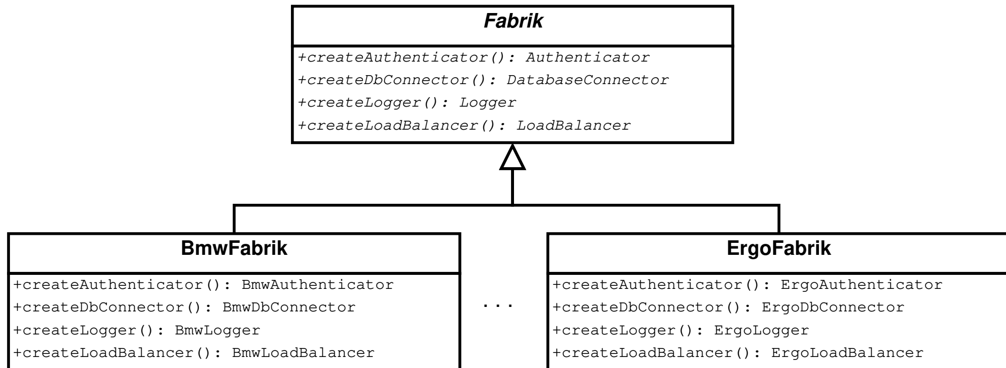
In den einzelnen Klassen unserer Software könnte dann stehen:

```
DatabaseConnector *dbCon;

if (kunde == "BMW")
    dbCon = new BmwDbConnector();
else if (kunde == "Ford")
    dbCon = new FordDbConnector();
.....
else if (kunde == "Provinzial")
    dbCon = new ProvinzialDbConnector();
else dbCon = new DefaultDbConnector();

dbCon->connect();
```

Guter Entwurf?

Problem: Kommt ein neuer Kunde hinzu, muss an vielen Stellen im Programm der Code ergänzt werden!

- Verlagere die Erzeugung der Objekte in eine eigene Klasse, in die Fabrik.
- Erstelle für jeden Kunden eigene Fabrik, die alle nötigen Objekte erzeugen kann.



Vorteile: Kommt ein neuer Kunde hinzu, muss nur

- eine neue Fabrik erstellt werden.
- für jede Basisklasse (`DatabaseConnector`, `Authenticator`, `Logger` usw.) eine abgeleitete Klasse erstellt oder die Default-Klasse verwendet werden.
- eine einzige Programmstelle geändert werden (bei Java mit Reflection-API nicht einmal das).

Durch die abstrakte Fabrik wird Konsistenz sichergestellt: An allen Stellen des Programms werden die Komponenten desselben Kunden verwendet.

erstellen der Fabrik:

```
Fabrik *fabrik;                  // globale Variable!

if (kunde == "BMW")
    fabrik = new BmwFabrik(...);
else if (kunde == "Opel")
    fabrik = new OpelFabrik(...);
...
else if (kunde == "Allianz")
    fabrik = new AllianzFabrik(...);
else fabrik = new DefaultFabrik(...);
```

aufrufen einer `create`*-Methode:*

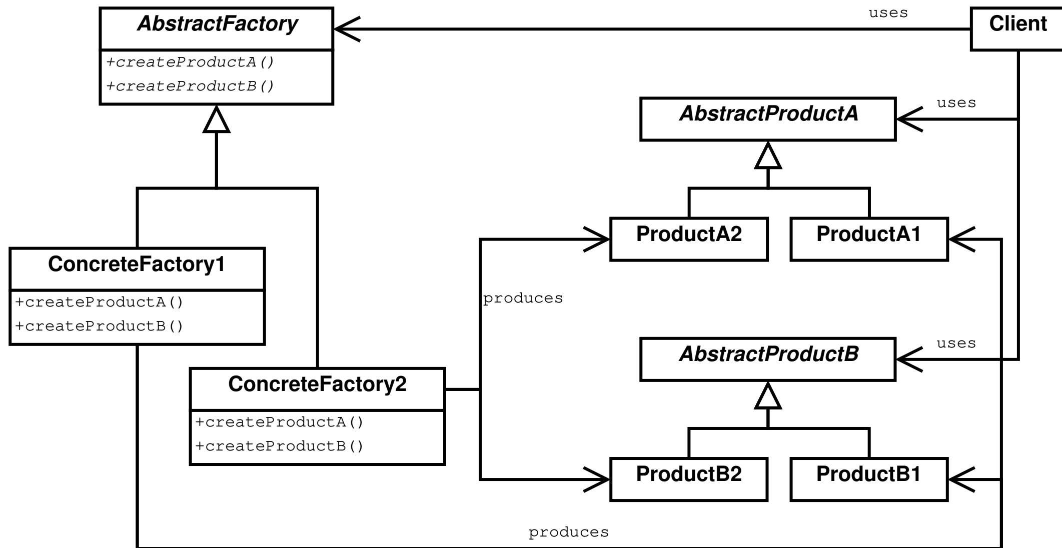
```
DatabaseConnector *dbCon;
dbCon = fabrik->createDbConnector(...);
dbCon->connect(...);
```

erstellen der Fabrik in Java mit Reflection-API:

```
Fabrik fabrik;
try {
    fabrik = Class.forName(System.getProperty("Fabriktyp"));
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    fabrik = new DefaultFabrik();
}
```

in Properties-Datei:

```
...
Fabriktyp = de.mycompany.fabrik.BmwFabrik
...
```


AbstractFactory
+createProductA()
+createProductB()
uses
Client
AbstractProductA
uses
ProductA2
ProductA1
ConcreteFactory1
+createProductA()
+createProductB()
produces
AbstractProductB
uses
ConcreteFactory2
+createProductA()
+createProductB()
ProductB2
ProductB1
produces

Nachteil:

- Die Fabrik muss an vielen Stellen, also in vielen Klassen bekanntgegeben werden.
- In C++ können wir mit globalen Variablen arbeiten, aber in anderen Programmiersprachen wie Java gibt es keine globalen Variablen.

Frage: Wie können wir sicherstellen, dass es

- nur eine Fabrik gibt?!
- nur einen `DatabaseConnector` gibt?!

→ Entwurfsmuster Singleton

Entwurfsmuster

Erzeuger und Strukturen

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

Verhalten

- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

Singleton

Stelle sicher, dass zu einer Klasse nur genau ein Objekt angelegt wird!

Anwendungen:

- Es gibt in einem System möglicherweise viele
  - Drucker, aber nur einen Drucker-Spooler.
  - Threads, aber nur einen Thread-Pool.
- Es gibt im System nur
  - eine Datenbankverbindung.
  - ein Objekt für das Logging.
  - einen Cache.
  - ein Objekt `DatumFormatDE`.

Singleton

Idee:

- Damit wir keine Objekte einer Klasse erzeugen können, muss der Konstruktor der Klasse `private` sein.

  In C++ müssen wir zusätzlich den Zuweisungsoperator und den Copy-Konstruktor ausschalten.
- Die statische Methode `exemplar` liefert das einzige Objekt der Klasse.

Implementierung:

```
class BmwFabrik: public Fabrik {
private:
    BmwFabrik();          // Konstruktor !!!
    static BmwFabrik *fabrik;
    ...

    // Zuweisungsoperator ausschalten
    BmwFabrik & operator=(BmwFabrik &);

    // Copy-Konstruktor ausschalten
    BmwFabrik(const BmwFabrik &);

public:
    static BmwFabrik *exemplar();   // !!!
    ...
};
```

```
BmwFabrik * BmwFabrik::fabrik = 0;

BmwFabrik * BmwFabrik::exemplar() {
    if (BmwFabrik::fabrik == 0)
        BmwFabrik::fabrik = new BmwFabrik(...);

    return BmwFabrik::fabrik;
}
```

Dadurch ist zwar sichergestellt, dass es nur eine einzige `BmwFabrik` gibt, aber unser Ausgangsproblem ist so noch nicht gelöst: Es kann immer noch verschiedene Fabriken geben.

- eine Stelle: `fabrik = OpelFabrik::exemplar();`
- andere Stelle: `fabrik = VwFabrik::exemplar();`

→ keine Konsistenzsicherung!

Singleton

Die globale Variable `fabrik` kann beliebig überschrieben werden!

- Verlagere die globale Variable `fabrik` als Attribut in die Oberklasse `Fabrik`.
- Wandle die Oberklasse `Fabrik` in ein Singleton.

```
class Fabrik {
private:
    static Fabrik *fabrik;

protected:
    Fabrik();                          // warum nicht private?
    Fabrik & operator=(Fabrik &);   // Zuweisungsop. aus
    Fabrik(const Fabrik &);          // Copy-Konstruktor aus

public:
    static Fabrik *exemplar();
};
```

Singleton

```
Fabrik * Fabrik::fabrik = 0;

Fabrik * Fabrik::exemplar() {
    if (Fabrik::fabrik == 0) {
        if (kunde == "BMW")
            Fabrik::fabrik = new BmwFabrik();
        else if (kunde == "Ford")
            Fabrik::fabrik = new FordFabrik();
        ...
        else Fabrik::fabrik = new DefaultFabrik();
    }

    return Fabrik::fabrik;
}
```

Problem: Für jeden neuen Kunden muss die bestehende Klasse geändert werden! Die Oberklasse muss alle Unterklassen kennen, auch die, die erst in Zukunft hinzugefügt werden!

Singleton

Lösung:

- Jede konkrete Fabrik registriert sich bei der `Fabrik`.
- Dazu erweitern wir die `Fabrik` um eine Liste von Fabriken.
- Jede Fabrik hat eine Methode `type`, die einen String zur Identifizierung (z.B. den Kundennamen) liefert. siehe dazu auch noch einmal das Entwurfsmuster Akteur-Rolle

```
class Fabrik {
private:
    static Fabrik *fabrik;
    static list<Fabrik *> fabriken;      // neu!
    ...
protected:
    Fabrik();
```

```
public:
    static Fabrik *exemplar();
    static void registerIt(Fabrik *);    // neu!
    virtual string type();               // neu!

    // eigentliche Schnittstelle definieren
    virtual Authenticator *createAuthenticator() = 0;

    virtual DatabaseConnector *createDbConnector() = 0;

    virtual LoadBalancer *createLoadBalancer() = 0;

    virtual Logger *createLogger() = 0;
};
```

```
// statische Variablen initialisieren
Fabrik * Fabrik::fabrik = 0;
list<Fabrik *> Fabrik::fabriken;          // neu!

string Fabrik::type() {                   // neu!
    return "Fabrik";
}

void Fabrik::registerIt(Fabrik *f) {    // neu!
    Fabrik::fabriken.push_back(f);
    cout << f->type() << " registered\n";
}
```

```
Fabrik * Fabrik::exemplar() {
    if (Fabrik::fabrik == 0) {
        list<Fabrik *>::iterator iter;

        iter = Fabrik::fabriken.begin();
        while (iter != Fabrik::fabriken.end()
                  && kunde != (*iter)->type()) {
            iter++;
        }

        if (iter == Fabrik::fabriken.end())
            Fabrik::fabrik = DefaultFabrik::exemplar();
        else Fabrik::fabrik = *iter;
    }

    return Fabrik::fabrik;
}
```

```
class VwFabrik : public Fabrik {
private:
    static VwFabrik *fabrik;
    string type() {                             // neu!
        return "VW";
    }
...
};

VwFabrik * VwFabrik::exemplar() {
    if (VwFabrik::fabrik == 0) {
        VwFabrik::fabrik = new VwFabrik();

        Fabrik::registerIt(VwFabrik::fabrik);   // neu!
    }
    return VwFabrik::fabrik;
}
```

Nachteil: Alle Fabriken müssen erzeugt werden und sich registrieren, obwohl nur eine Fabrik benötigt wird!

Frage: Wer veranlasst eigentlich die Erzeugung der konkreten Fabriken, damit sie sich registrieren können?

Die Oberklasse `Fabrik` kann das nicht, da es die konkreten Klassen nicht kennt - und auch nicht kennen soll!

Antwort: Wir müssen direkt ein statisches Objekt anlegen.

```
class VwFabrik : public Fabrik {
private:
    static VwFabrik fabrik;   // kein Zeiger!
    VwFabrik();
...
};

VwFabrik VwFabrik::fabrik;    // Objekt anlegen!

VwFabrik::VwFabrik() {
    Fabrik::registerIt(this); // neu im Konstr.!
}

// die Methode "exemplar()" gibt es nicht mehr

...
```

Entwurfsmuster

Erzeuger und Strukturen

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

Verhalten

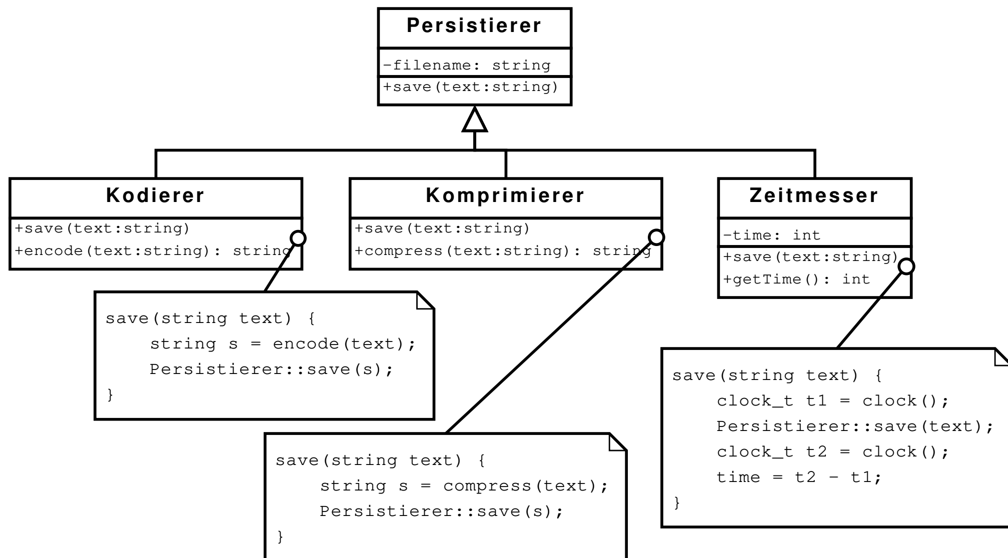
- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

Motivation: Unsere Software hat eine Klasse `Persistierer`, die dazu dient, einen beliebigen Text in einer Datei abzuspeichern.

Im Laufe der Zeit kommen weitere Anforderungen an den Persistierer hinzu:

- Der Text soll kodiert abgelegt werden.
- Der Text soll komprimiert werden.
- Zu Testzwecken soll die zum Abspeichern benötigte Zeit gemessen werden.

mögliche Lösung:

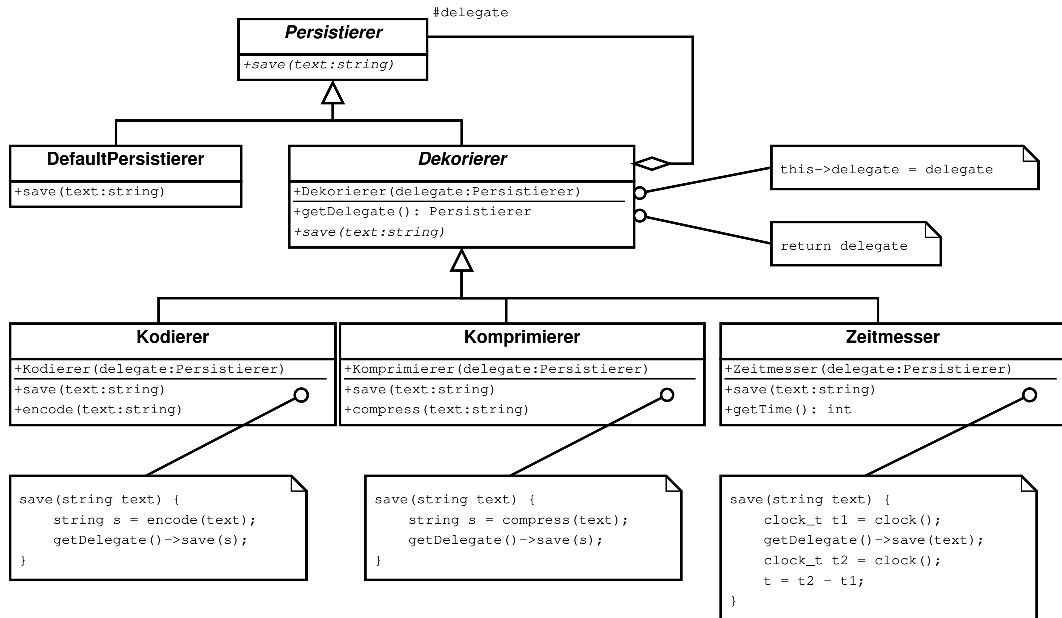


Decorator

Problem: Wie kombiniert man die Unterklassen?

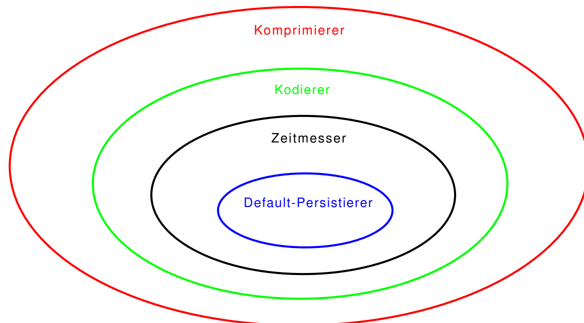
- Wir wollen einen Text zunächst kodieren und dann komprimieren.
- Während der Testphase wollen wir wissen, wie lange das Komprimieren dauert.

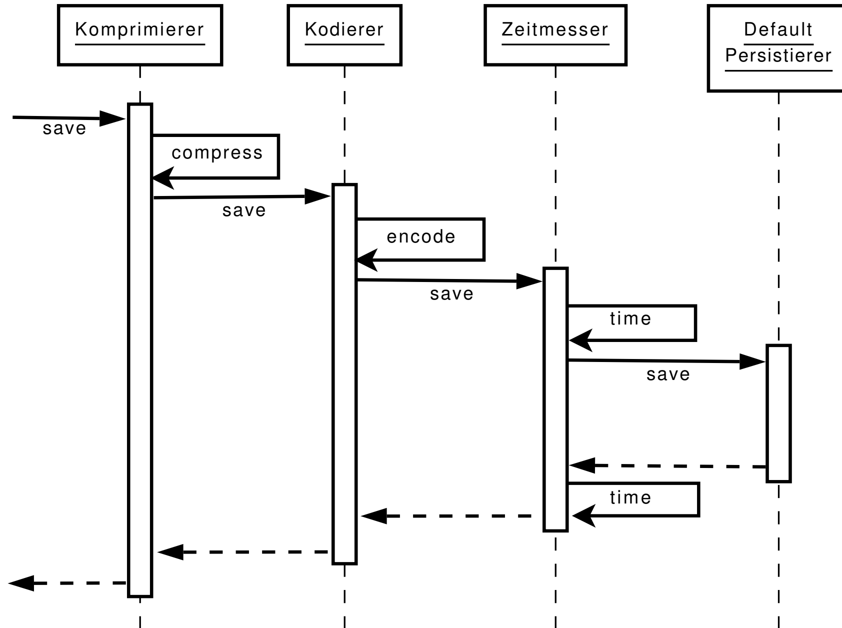
Lösung: Verwende Delegation anstelle von Vererbung!


#delegate
Persistierer
+save(text:string)
DefaultPersistierer
+save(text:string)
Dekorierer
+Dekorierer(delegate:Persistierer)
+getDelegate(): Persistierer
+save(text:string)
this->delegate = delegate
return delegate
Kodierer
+Kodierer(delegate:Persistierer)
+save(text:string)
+encode(text:string)
Komprimierer
+Komprimierer(delegate:Persistierer)
+save(text:string)
+compress(text:string)
Zeitmesser
+Zeitmesser(delegate:Persistierer)
+save(text:string)
+getTime(): int
save(string text) {
string s = encode(text);
getDelegate()->save(s);
}
save(string text) {
string s = compress(text);
getDelegate()->save(s);
}
save(string text) {
clock_t t1 = clock();
getDelegate()->save(text);
clock_t t2 = clock();
t = t2 - t1;
}

Wie wird es verwendet?

```
string text("Dies ist nur ein kleiner Test");
Persistierer *p = new Komprimierer(
    new Kodierer(
        new Zeitmesser(
            new DefaultPersistierer("test.txt"))));
p->save(text);
```



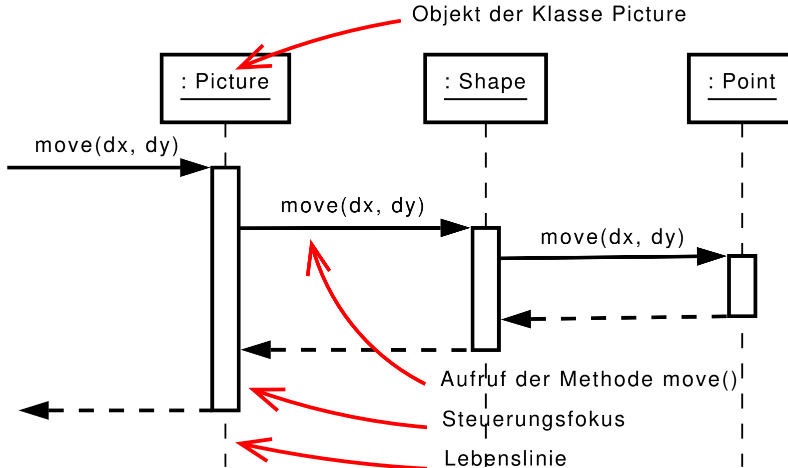


Komprimierer
Kodierer
Zeitmesser
Default Persistierer
save
compress
save
encode
save
time
save
time

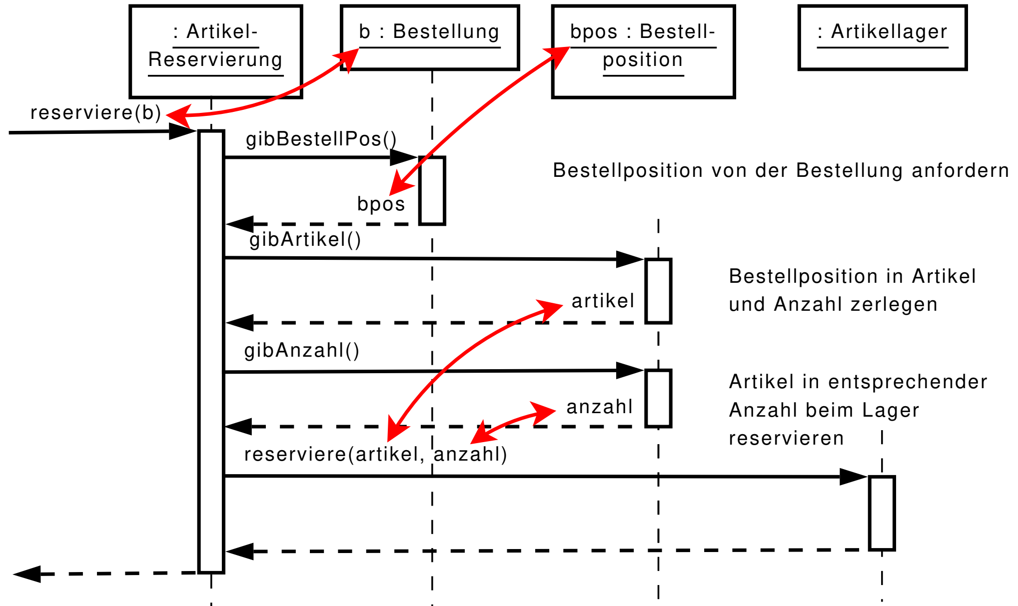
```java
import java.io.*;
import java.util.zip.*;

public class StreamTest {
    public static void main(String[] args) {
        try {
            String str = new String(".....");

            // Ausgabe in eine Datei
            ObjectOutputStream out =
                new ObjectOutputStream(
                    new GZIPOutputStream(
                        new FileOutputStream("abc.zip")));
            out.writeObject(str);
            out.flush();
            out.close();
            .....
```

```java
        // Einlesen aus einer Datei
        ObjectInputStream in =
           new ObjectInputStream(
              new GZIPInputStream(
                 new FileInputStream("abc.zip")));
        System.out.println(in.readObject());
     } catch (Exception e) {
        e.printStackTrace();
     }
  }
}
```

Ein *Sequenzdiagramm* zeigt eine Reihe von Nachrichten, die eine ausgewählte Menge von Objekten in einer zeitlich begrenzten Situation austauscht, wobei der zeitliche Ablauf betont wird.



Einschub: UML Sequenzdiagramme



Erzeuger und Strukturen

- Abstrakte Fabrik (Abstract Factory)
- Einzelstück (Singleton)
- Dekorierer (Decorator)
- Kompositum (Composite)

Verhalten

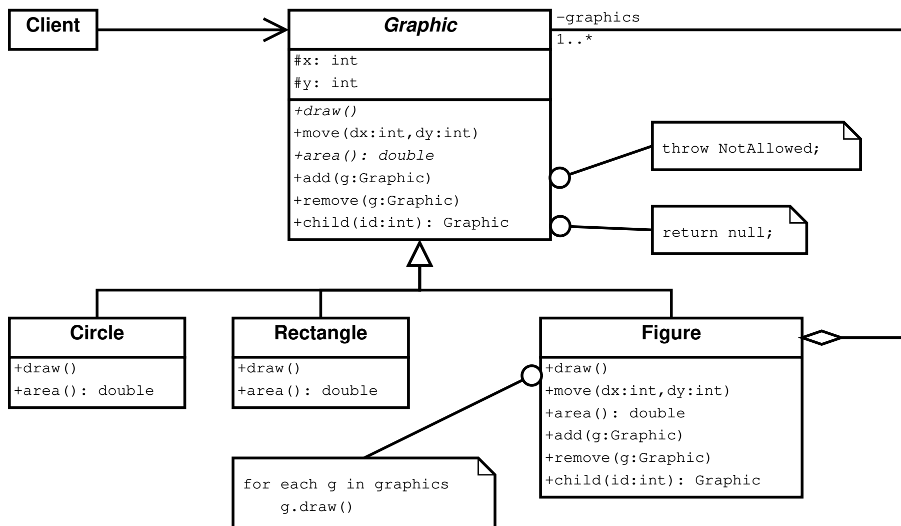
- Strategie (Strategy)
- Zustand (State)
- Beobachter (Observer)
- Model-View-Controller

Motivation

Oft ist es in grafischen Anwendungen wie Zeicheneditoren möglich, komplexe Diagramme aus einfachen Komponenten aufzubauen.

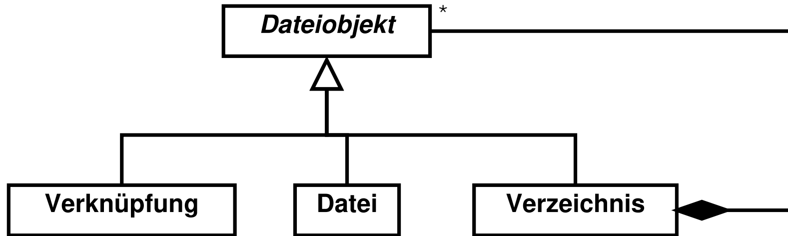
- Als Primitive stehen oft Komponenten wie Rechteck, Kreis, Dreieck oder allgemein geschlossene Polygonzüge zur Verfügung.
- Der Benutzer kann Komponenten zu größeren Komponenten zusammenfassen, die wiederum zu noch größeren Komponenten zusammengefasst werden können usw.
- Alle diese Komponenten können in vielen Fällen gleich behandelt werden. Alle haben eine
  - `move()`-Methode, mit der sich das Objekt verschieben lässt.
  - `area()`-Methode, die den Flächeninhalt des Objekts liefert.
  - `draw()`-Methode, die das Objekt zeichnet.
- Wie können wir erreichen, dass der Klient sowohl primitive als auch zusammengesetzte Objekte einheitlich behandeln kann?

Composite



Die Klasse `Figure` definiert ein Aggregat, also eine Zusammenfassung von `Graphic`-Objekten. Die Methode `draw` ruft einfach die draw-Methode der Kindobjekte auf.

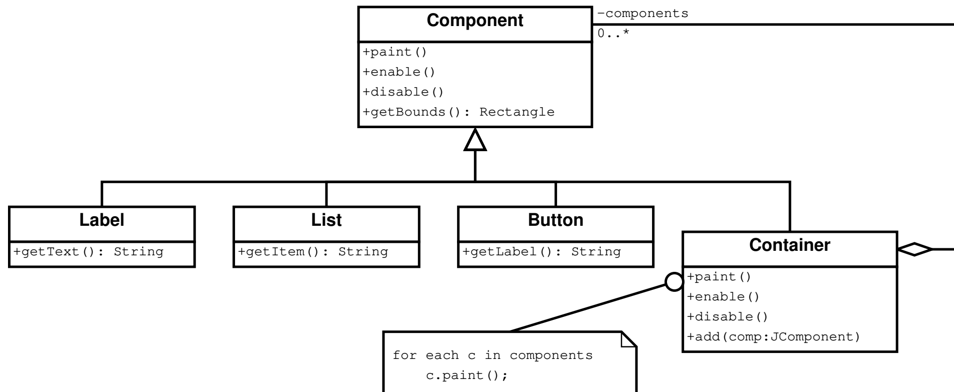
Ein ähnliches Diagramm hatten wir schon einmal im Abschnitt „Komposition“. Nur wussten wir damals noch nicht, dass dies ein Entwurfsmuster ist.



Ein Verzeichnis ist eine Zusammenfassung von Dateiobjekten. Primitive Dateiobjekte sind echte Dateien und Verweise auf Dateien. Wird ein Verzeichnis gelöscht, werden auch alle darin enthaltenen Dateiobjekte gelöscht.

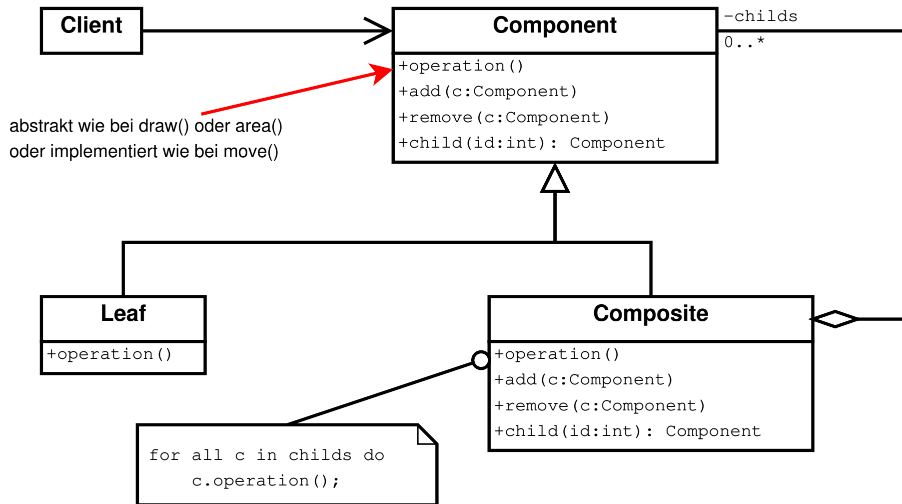
Composite

Benutzeroberflächen bauen Fenster aus primitiven Objekten wie Label, List oder Button und aus zusammengesetzten Objekten (Container) auf.



Alle Komponenten haben eine Methode zum Zeichnen des Objekts, Methoden zum Aktivieren und Deaktivieren des Objekts usw., die in der Basisklasse implementiert sind. Spezielle Methoden sind in speziellen Klassen implementiert. (design for type safety)

Grundlegende Struktur des Kompositum-Musters (design for uniformity):

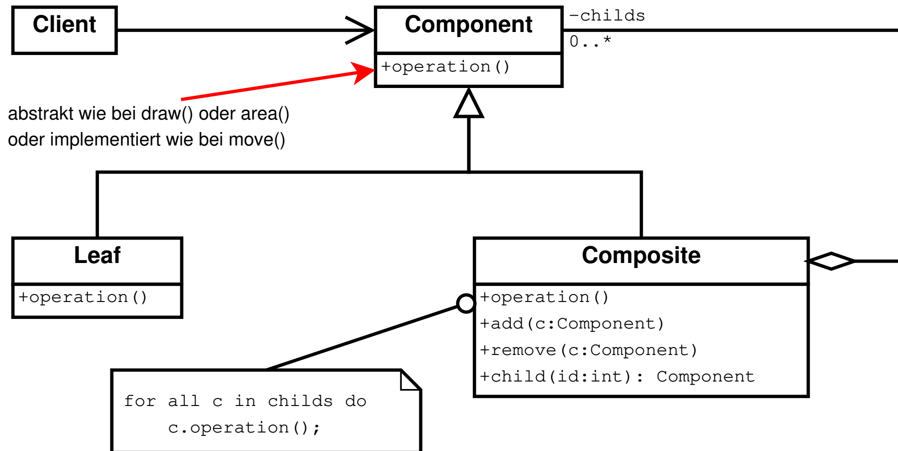


Anmerkungen:

Die allgemeine Basisklasse `Component` bietet eine Standard-Implementierung, die ggf. von den Blatt- und Kompositionsknoten überschrieben wird. Im Fall des Zeicheneditors:

- In der Klasse `Graphic` implementieren wir die Methoden wie folgt:
  - `child(int id)` liefert `NULL`.
  - `add(Graphic)` wirft eine Exception.
  - `remove(Graphic)` wirft eine Exception.
- Die Kompositionsklasse `Figure` leitet die Aufrufe der Methoden `draw`, `move` und `area` an die einzelnen Graphic-Objekte weiter und überschreibt die Methoden `add`, `remove` und `child`.

Grundlegende Struktur des Kompositum-Musters (design for type safety):



Composite

There are two design variants for defining and implementing child-related operations like adding/removing a child component to/from the container and accessing a child component:

- *Design for uniformity:* Child-related operations are defined in the `Component` interface. This enables clients to treat `Leaf` and `Composite` objects uniformly. But type safety is lost because clients can perform child-related operations on `Leaf` objects.
- *Design for type safety:* Child-related operations are defined only in the `Composite` class. Clients must treat `Leaf` and `Composite` objects differently. But type safety is gained because clients cannot perform child-related operations on `Leaf` objects.

Quelle: `https://en.wikipedia.org/wiki/Composite_pattern`

Anmerkungen:

- Werden explizit Referenzen auf Eltern-Objekte gespeichert, kann dies die Traversierung der Kompositionsstruktur vereinfachen.

  Sinnvoll ist es, diese Referenz in Component zu speichern, da alle abgeleiteten Klassen dieses Attribut erben.
- Der Behälter zur Speicherung der enthaltenen Kinder wie bspw. eine Liste wird im Composite abgelegt, nicht in Component. Ansonsten würde z.B. ein Rechteck einen Behälter zur Speicherung von Kindern bereit stellen, obwohl gar keine Kinder gespeichert werden. $\rightarrow$ Verschwendung von Speicherplatz.

  Die Art des Behälters, also bspw. Liste, Dictionary oder Hash-Map beeinflusst – wie immer – die Laufzeit der Operationen.
- Das Löschen eines Kompositums hat in der Regel zur Folge, dass auch die enthaltenen Kindobjekte gelöscht werden. Diese Aufgabe sollte im Destruktor des Kompositums erledigt werden.